.NET FULL STACK Development Program

—

Day 2 C# Basic

Outline

- Comment
- Variable and Data Type
- Operator
- Flow Control
- Keywords

Comment

In C#, there are 3 types of comments:

- Single-line Comments (//)
  Single-line comments start with a double slash //. The compiler ignores everything after // to the end of the line.

- Multi-line Comments (/* */)
  Multi-line comments start with /* and ends with */. Multi-line comments can span over multiple lines.

- XML Comments (///)
  XML documentation comment is a special feature in C#. It starts with a triple slash /// and is used to categorically describe a piece of code.. This is done using XML tags within a comment. These comments are then, used to create a separate XML documentation file.

```
int myInt = 100 + 1; // result is 101
```

```
/*
 * This is a multi-line comment
 * This is a multi-line comment
 * This is a multi-line comment
 */
```

```
/// <summary>
/// This is the Main method
/// The entry point of a C# application
/// </summary>
0 references
static void Main(string[] args)
{
```

Variable and Data Type

Variable

- A variable is a named memory location that we can create in order to store data.
- Every variable has a type that determines what values can be stored in the variables.

- Declaration
  [data type] [variable name];
- Initialization
  [variable name] = [value];
- Combined
  [data type] [variable name] = [value];

```C#
//declare the variable age first, initialize it later
int age;
age = 18;

//declare the variable and assign a value
int fullScore = 100;
```

Variables and Data Types

- Types of variables
  - A data type specifies the type of data that a variable can store.
  - Two kinds of types:
    - Value types:
      simple types, *enum types*, *struct types*, *nullable value types*, and *tuple value types.*
    - *Reference types:*
      string, class types, *interface types*, *array types*, and *delegate types.*

Value Type

- Simple Types
  - Signed integral: sbyte, short, int, long
  - Unsigned integral: byte, ushort, uint, ulong
  - floating-point: float, double
  - High-precision decimal floating-point: decimal
  - Unicode characters: char
  - Boolean: bool

Signed and Unsigned Integral

C# supports the following predefined integral types:

C# type/keyword	Range	Size	.NET type
`sbyte`	-128 to 127	Signed 8-bit integer	System.SByte
`byte`	0 to 255	Unsigned 8-bit integer	System.Byte
`short`	-32,768 to 32,767	Signed 16-bit integer	System.Int16
`ushort`	0 to 65,535	Unsigned 16-bit integer	System.UInt16
`int`	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	System.Int32
`uint`	0 to 4,294,967,295	Unsigned 32-bit integer	System.UInt32
`long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	System.Int64
`ulong`	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	System.UInt64

nint & nuint

- Starting in C# 9.0, you can use the nint and nuint keywords to define ***native-sized integers***. These are 32-bit integers when running in a 32-bit process, or 64-bit integers when running in a 64-bit process.
- The native-sized integer types are represented internally as the .NET types System.IntPtr and System.UIntPtr.
- Range:
  - For **nint**: Int32.MinValue to Int32.MaxValue.
  - For **nuint**: UInt32.MinValue to UInt32.MaxValue.
- There’s no direct syntax for native-sized integer literals. There’s no suffix to indicate that a literal is a native-sized integer, such as L/l to indicate a long. You can use implicit or explicit casts of other integer values instead.

```
nint nativeSignedInt = 10;
nuint nativeUnsignedInt = (nuint)11;
Console.WriteLine(nativeSignedInt.GetType()); // System.IntPtr
Console.WriteLine(nativeUnsignedInt.GetType());//System.UIntPtr
```

Floating-point numeric types

C# type/keyword	Approximate range	Precision	Size	.NET type
`float`	±1.5 x 10^{-45} to ±3.4 x 10^{38}	~6-9 digits	4 bytes	System.Single
`double`	±5.0 × 10^{-324} to ±1.7 × 10^{308}	~15-17 digits	8 bytes	System.Double
`decimal`	±1.0 x 10^{-28} to ±7.9228 x 10^{28}	28-29 digits	16 bytes	System.Decimal

```
//to declare floating-point numbers
float myFloat = 11.1f;
double myDouble = 11.1;
decimal myDecimal = 11.1m;
```

Unicode Character: char

The **char** type keyword represents a Unicode UTF-16 character.

Type	Range	Size	.NET type
`char`	U+0000 to U+FFFF	16 bit	System.Char

```
char myChar = 'a';
myChar = '@';
```

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

Type Conversion and Casting

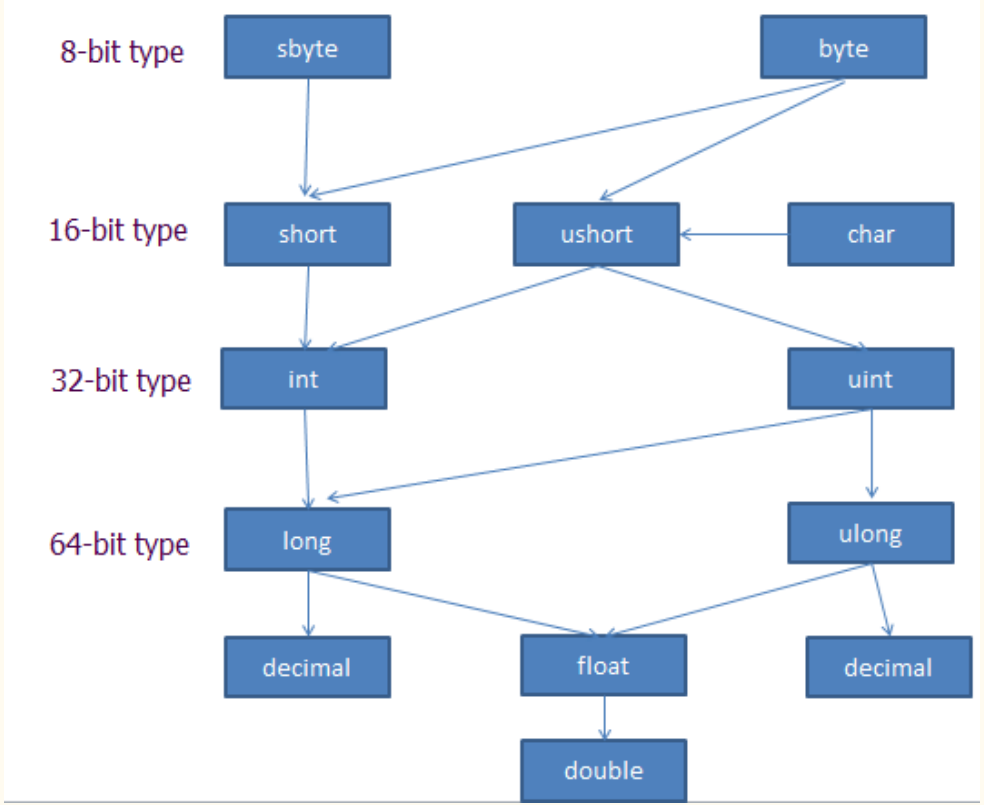
- **Implicit conversions**: No special syntax is required because the conversion always succeeds, and "no data will be lost". Examples include conversions from smaller to larger integral types.

  - **Implicit Casting** (automatically) - converting a smaller type to a larger type size
    `char` -> `int` -> `long` -> `float` -> `double`

- **Explicit conversions (casting)**: Explicit conversions require a cast expression. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range.

  - **Explicit Casting** (manually) - converting a larger type to a smaller size type
    `double` -> `float` -> `long` -> `int` -> `char`

Type Conversion and Casting



Boolean

The **bool** type keyword represents a Boolean value, which can be either true or false.

bool	1 bit	Stores true or false values

C#

```
bool itemAvailability = true;

itemAvailability = false;
```

Value Types

- Value types
  - Simple types
    - Signed integral: sbyte, short, int, long
    - Unsigned integral: byte, ushort, uint, ulong
    - floating-point: float, double
    - High-precision decimal floating-point: decimal
    - Unicode characters: char
    - Boolean: bool
  - Enum types
    - User-defined types of the form enum E {...}. An enum type is a distinct type with named constants. Every enum type has an underlying type, which must be one of the eight integral types. The set of values of an enum type is the same as the set of values of the underlying type.
  - Struct types
    - User-defined types of the form struct S {...}
  - Nullable value types
    - Extensions of all other value types with a null value
  - Tuple value types
    - User-defined types of the form (T1, T2, ...)

Enumeration type

An ***enumeration type*** (or *enum type*) is a value type. To define an enumeration type, use the ***enum*** keyword and specify the names of *enum members*

```
enum Week
{
    Monday,    // 0
    Tuesday,   // 1
    Wednesday, // 2
    Thursday,  // 3
    Friday,    // 4
    Saturday,  // 5
    Sunday     // 6
}
0 references
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
    //an enum value can be access by using the [enum_name].[member_name]
    Week week = Week.Monday;
    Console.WriteLine(week);

    //we can also access thru the index
    int enumIndex = (int)Week.Sunday;
    Console.WriteLine(enumIndex); // 6

    Console.WriteLine((Week)4); // Friday
}
```

Structure type

- A structure type (or struct type) is a value type that can encapsulate data and related functionality. You use the struct keyword to define a structure type.
- We can have variables, methods, and constructors defined in the struct.
- Struct doesn’t allow no-arg constructors.

```
3 references
struct Album
{
    string _title;
    string _singer;
    string _date;

    1 reference
    public Album(string title, string singer, string date)
    {
        _title = title;
        _singer = singer;
        _date = date;
    }

    1 reference
    public void GetReleaseDate ()
    {
        Console.WriteLine(_date);
    }
}
0 references
static void Main(string[] args)
{
    // we need to use new keyword to create an album instance
    Album album = new Album("Recovery", "Eminem", "2010-06-18");
    //call the method in the struct and write the release date in the console
    album.GetReleaseDate(); // 2010-06-18
}
```

Tuple type

- Available in C# 7.0 and later, tuple types are value types, the tuples feature provides concise syntax to group multiple data elements in a lightweight data structure.

```
(string, int) t1 = ("Liam", 22);
(string name, int age) t2 = ("Lily", 23);
Console.WriteLine("Name is " + t1.Item1 + ", " + "Age is " + t1.Item2);
Console.WriteLine($"Name is {t2.name}, Age is {t2.age}");
```

Nullable Value Type - ?

- Starting from C# 8.0, A nullable value type represents all values of its underlying value type and an additional null value. For example, you can assign any of the following three values to a ***bool?*** variable: **true**, **false**, or **null**.

```
int? num = null;
int num2 = num; // cannot compile

char? character = 'a';
character = null;
```

- The null-coalescing operator ?? returns the value of its left-hand operand if it isn't null; otherwise, it evaluates the right-hand operand and returns its result. The ?? Operator doesn't evaluate its right-hand operand if the left-hand operand evaluates to non-null.

```
int? a = 20;
int? b = a +10;
Console.WriteLine("a = " + a + " and b =" + b);
a = null;
b = a + 10 ?? -1; // if a + 10 is null, assign -1 to b
Console.WriteLine("a = " + a + " and b =" + b);
```

```
a = 20 and b =30
a =  and b =-1
```

Reference Types

- C# provides the following built-in reference types:
  - string
  - dynamic
  - object

- The following keywords are used to declare reference types:
  - class
  - delegate
  - interface
  - record

String type

The **string** type represents a sequence of zero or more Unicode characters (char).

```
string sayHi = "Hello December!";
//we can use index to access every char in the string
char oneChar = sayHi[0];
Console.WriteLine(oneChar); // H
```

Immutability of Strings

String objects are ***immutable***: they can't be changed after they've been created.

When we assign a value to a string or modify the string, C# will create a new string.

```
Console.WriteLine("String Immutability:");
string herName = "Sophia";
herName.ToUpper(); // Cap all the letters in the string
Console.WriteLine(herName);
```

String Interning

- The common language runtime(CLR) conserves string storage by maintaining a table, called the ***intern pool***, that contains a single reference to each unique literal string declared or created programmatically in your program. Consequently, an instance of a literal string with a particular value only exists once in the system.

- For example, if you assign the same literal string to several variables, the runtime retrieves the same reference to the literal string from the intern pool and assigns it to each variable.

String Concatenation

```
// string concatenation -> +
string str1 = "New";
string str2 = "Jersey";
string combinedString1 = str1 + str2;
Console.WriteLine(combinedString1); // NewJersey

//string concatenation -> Concat()
string combinedString2 = string.Concat(str1, str2);
Console.WriteLine(combinedString2); // NewJersey

//string interpolation
string stringInterpolation = $"{combinedString2} is the garden state.";
Console.WriteLine(stringInterpolation);

//string formatting
(string name, int age) myInfo = ("Liam", 22);
Console.WriteLine("Hello, I am {0}, {1} years old.", myInfo.name, myInfo.age);
```

- string.Format();
- string.Join();
- Append() method in StringBuilder;

Class

A class is a container that contains the block of code that includes field, method, constructor, etc.

```
class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }

    // Constructor:
    public Child(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Printing method:
    public void PrintChild()
    {
        Console.WriteLine("{0}, {1} years old.", name, age);
    }
}
```

Delegate

A **delegate** is a reference type that can be used to encapsulate a named or an anonymous method. The delegate must be instantiated with a method or lambda expression that has a compatible return type and input parameters.

```
//Define Delegate
public delegate void Hello();

0 references
public static void Main(string[] args)
{
    //approach 1
    Hello delegate1 = Function;
    //approach 2
    Hello delegate2 = delegate () { Console.Write("Hi2"); };
    //approach 3 - lambda
    Hello delegate3 = () => { Console.Write("Hi3"); };
    //approach 4
    Hello delegate4 = new Hello(Function);
    //approach 5
    Hello delegate5 = delegate1+ delegate2+ delegate3+ delegate4;

    //invoke delegate function
    delegate5();

}

2 references
public static void Function() { Console.Write("Hi"); }
```

Dynamic type

The **dynamic** type indicates that the use of the variable and references to its members **escapes compile-time type checking.** Instead, it resolves the type at run time.

The **dynamic** types change types at run-time based on the assigned value, so **the type dynamic only exists at compile-time, not run-time**.

The dynamic type variables are converted to other types implicitly.

```
Console.WriteLine("dynamic types change types at run-time!");

dynamic MyDynamicVar = 100;
Console.WriteLine("Value: {0}, Type: {1}", MyDynamicVar, MyDynamicVar.GetType());

MyDynamicVar = "Hello World!";
Console.WriteLine("Value: {0}, Type: {1}", MyDynamicVar, MyDynamicVar.GetType());

MyDynamicVar = true;
Console.WriteLine("Value: {0}, Type: {1}", MyDynamicVar, MyDynamicVar.GetType());
```

```
dynamic d1 = 100;
int i = d1;

d1 = "Hello";
string greet = d1;
```

Other Data Types

Implicit type - Var

Beginning with C# 3, variables that are declared at method scope can have an implicit "type" **var**. An implicitly typed local variable is strongly typed just as if you had declared the type yourself, but the compiler determines the type.

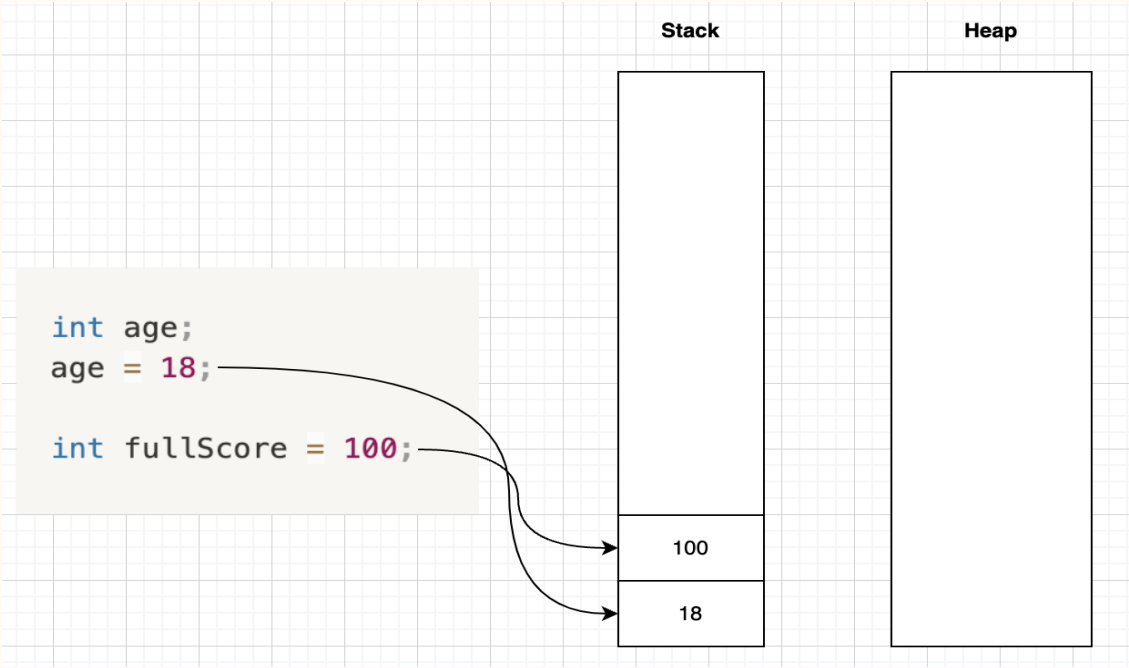
```C#
int age; //explicitly typed
age = 18;

var fullScore = 100; //implicitly typed

var month = "November"; //implicitly typed
```

Var vs. Dynamic

var vs. dynamic	var	dynamic
Type	var variables are identified at **compile time**.	dynamic variables are identified at **run time**.
Declaration	var variables **must be initialized** at the time of their declaration.	dynamic variables are **not mandatory to get initialized** at the time of declaration.
Value	var variables are **statically typed** and they cannot be assigned different data type values.	dynamic variables are **dynamically typed**, and they can be assigned different data type values.

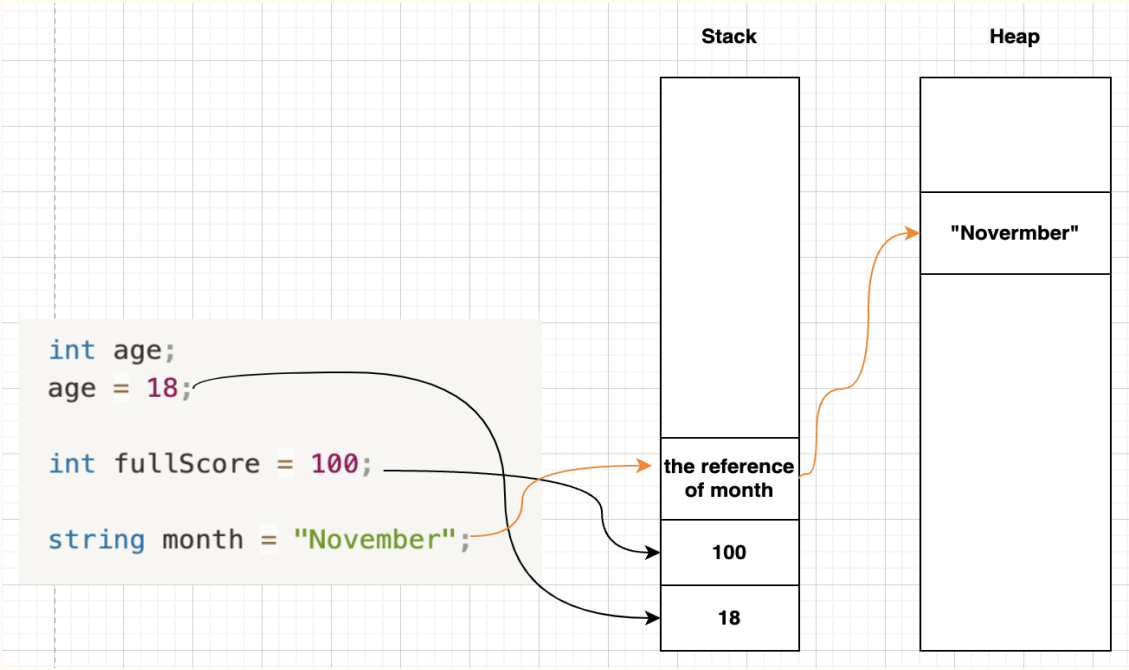
Stack

- Used for static memory allocation
- Contains data for value types
- Contains references to reference types(objects)
- Relatively small
- Access speed is fast



Heap

- Used for dynamic memory allocation
- Contains the data of reference types
- Relatively large
- Access speed is slow



Operator

Operator

Arithmetic operations in C#

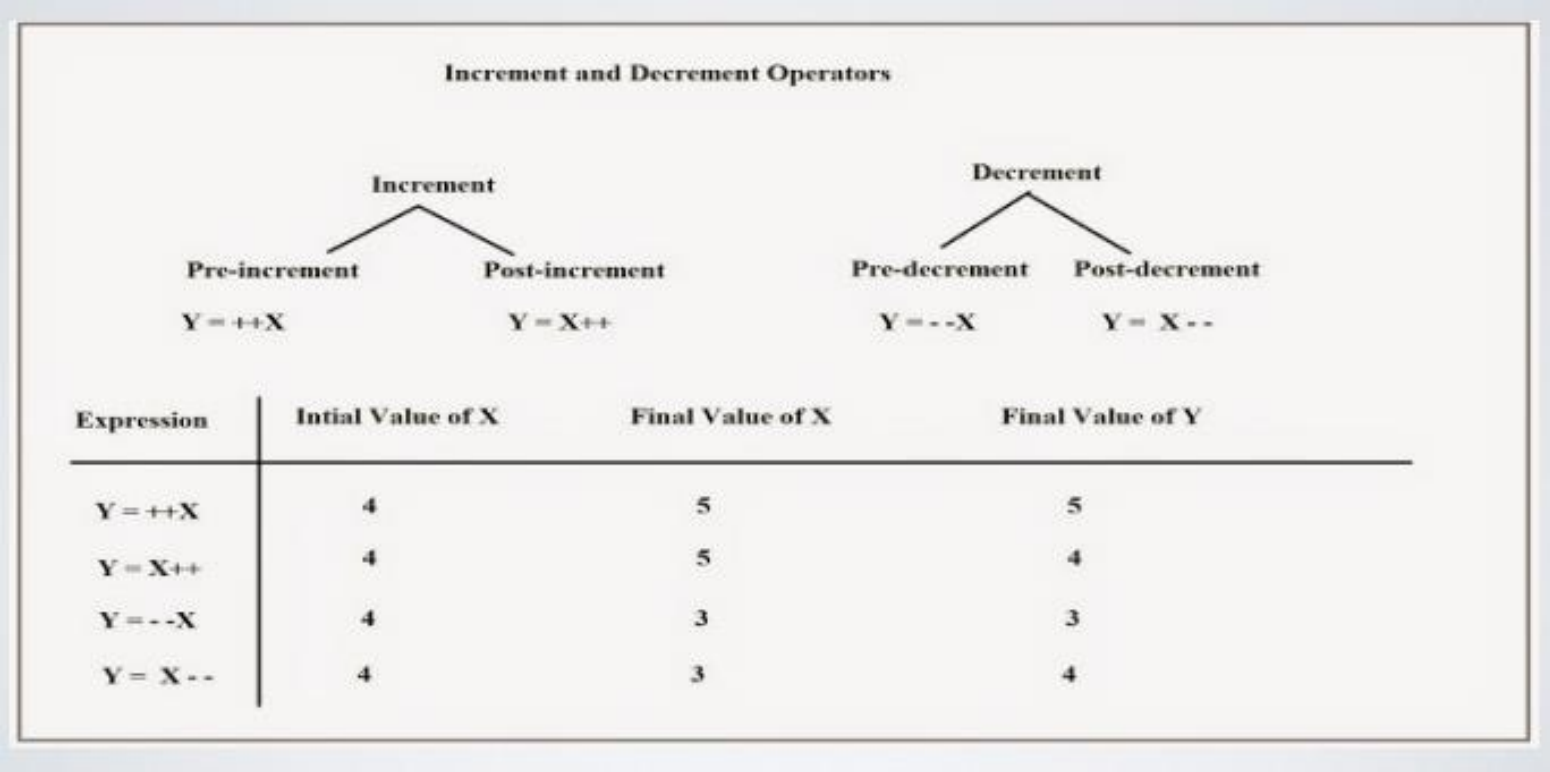
- Precedence: (*, /, %) > (+, -)
- Parentheses: evaluate the innermost parenthesized expression first, and work your way out through the levels of nesting
- No {} or [ ] in parentheses in C#

```C#
int x = 1, y = 2, z;

z = x + y * 2; // 5

z = (x + y) * 2 //6
```

Compound Arithmetic/ Assignment Operators

Operator	Use	Meaning
+=	`x += 1;`	`x = x + 1;`
-=	`x -= 1;`	`x = x - 1;`
*=	`x *= 5;`	`x = x * 5;`
/=	`x /= 2;`	`x = x / 2;`
%=	`x %= 10;`	`x = x % 10;`

Increment and Decrement Operators



Expression	Intial Value of X	Final Value of X	Final Value of Y
Y = ++X	4	5	5
Y = X++	4	5	4
Y = - -X	4	3	3
Y = X - -	4	3	4

Logical Operator

Operator	Name	Type	Description
!	Not	Unary	Returns true if the operand to the right evaluates to false. Returns false if the operand to the right is true.
&	And	Binary	Binary operation: logical and 0001 & 0011 = 0001 = 1
|	Or	Binary	Binary operation: logical or 0001 | 0011 = 0011 = 3
^	Xor	Binary	Binary Operation: exclusive or 0001 ^ 0011 = 0010 = 2
&&	Conditional And	Binary	If the operand on the left returns false, it returns false without evaluating the operand on the right.
||	Conditional Or	Binary	If the operand on the left returns true, it returns true without evaluating the operand on the right.

Relational Operators

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Flow Control

- Selection Statements
  - if, switch
- Iteration Statements
  - for, foreach, do-while, while
- Jump Statements
  - break, continue, return, goto

If

```
int time = 22;

if(time < 10)
{
    Console.WriteLine("Good Morning!");
}else if(time < 20) //when 10 <= time < 20
{
    Console.WriteLine("Good Day!");
}
else //when time >= 20
{
    Console.WriteLine("Good Evening!");
}
```

Ternary Operators

variable = condition ? expressionTrue : expressionFalse;

```
bool valid = true;
int i;

if(valid)
{
   i = 1;
} else
{
   i = 0;
}
```

```
//Ternary Operators
bool valid = true;
int i;
i = valid ? 1 : 0;
```

Switch

```
char letter = 'A';
switch (letter)
{
    case 'A':
        Console.WriteLine("Letter A");
            break;

    case 'B':
        Console.WriteLine("Letter B");
        break;

    default:
        Console.WriteLine("A Letter");
        break;
}
```

```
double measurement = 22.5;

switch (measurement)
{
    // (Introduced in C# 9.0)
    case < 0.0:
        Console.WriteLine($"Measured value is {measurement}; too low.");
        break;

    case > 15.0:
        Console.WriteLine($"Measured value is {measurement}; too high.");
        break;

    default:
        Console.WriteLine($"Measured value is {measurement}.");
        break;
}
```

Iteration Statement

- For
- Foreach
- While
- Do-while

For

```
for (initialization; condition; iterator)
{
  // statements to keep executing while condition is true
}
```

```
for (int i = 0; i < 3; i++)
{
    Console.Write(i);
}
// Output:
// 012
```

Foreach

```
foreach (element in iterable-item)
{
    // body of foreach loop
}
```

```
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
foreach (int element in fibNumbers)
{
    Console.Write($"{element} ");
}
// Output:
// 0 1 1 2 3 5 8 13
```

While

```
while (condition)
{
  // body of while
}
```

```
int n = 0;
while (n < 5)
{
    Console.Write(n);
    n++;
}
// Output:
// 01234
```

Do-while

```
do
{

  //body of do-while loop

} while (condition)
```

```
int n = 0;
do
{
    Console.Write(n);
    n++;
} while (n < 5);
// Output:
// 01234
```

Jump Statement

- Break
- Continue
- Return
- Goto

Break

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach (int number in numbers)
{
    if (number == 3)
    {
        break;
    }

    Console.Write($"{number} ");
}
Console.WriteLine();
Console.WriteLine("End of the example.");
// Output:
// 0 1 2
// End of the example.
```

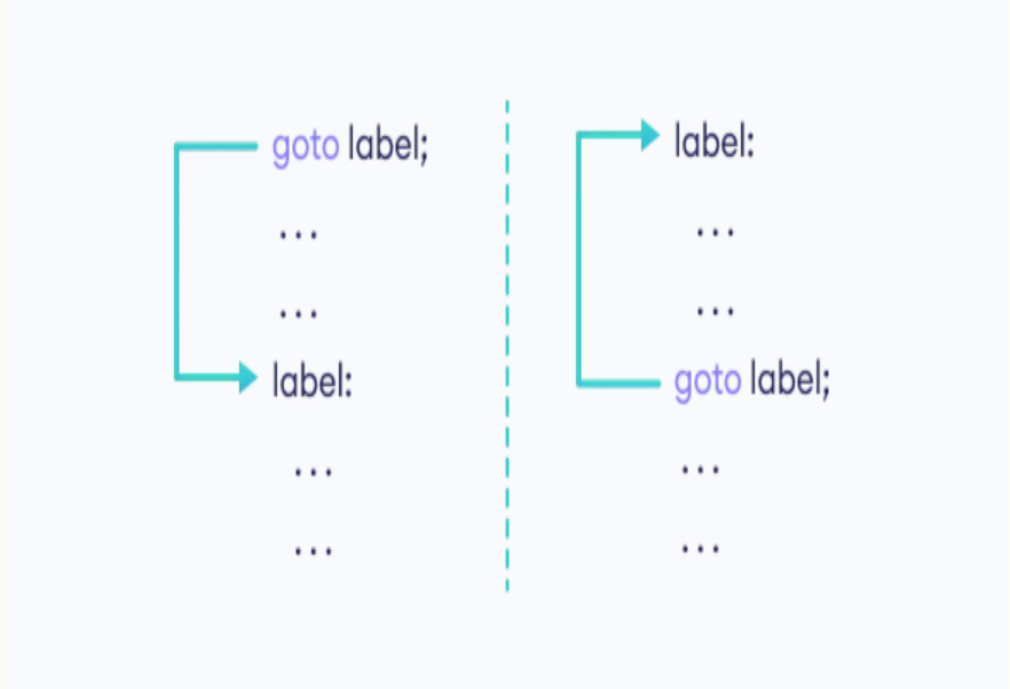
Continue

```
for (int i = 0; i < 5; i++)
{
    Console.Write($"Iteration {i}: ");

    if (i < 3)
    {
        Console.WriteLine("skip");
        continue;
    }

    Console.WriteLine("done");
}
// Output:
// Iteration 0: skip
// Iteration 1: skip
// Iteration 2: skip
// Iteration 3: done
// Iteration 4: done
```

Return

```
Console.WriteLine("First call:");
DisplayIfNecessary(6);

Console.WriteLine("Second call:");
DisplayIfNecessary(5);

void DisplayIfNecessary(int number)
{
    if (number % 2 == 0)
    {
        return;
    }

    Console.WriteLine(number);
}
// Output:
// First call:
// Second call:
// 5
```

Goto



Keywords

- Namespace
- Assembly
- Access Modifiers

Namespace

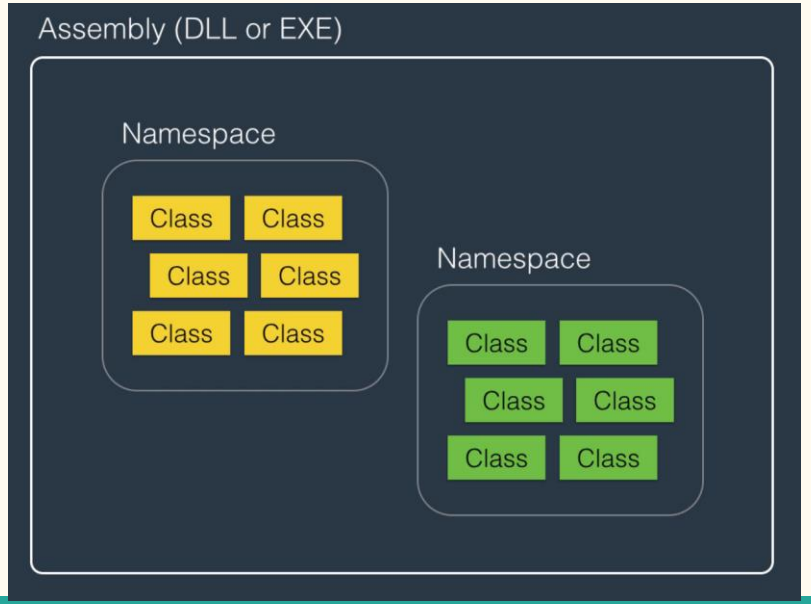
The **namespace** keyword is used to declare a scope that contains a set of related objects. You can use a namespace to organize code elements and to create globally unique types.

- You can think is as a package that includes classes that can be used elsewhere
- *namespace is often the file name

```
namespace SampleNamespace
{
    class SampleClass { }

    interface ISampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace Nested
    {
        class SampleClass2 { }
    }
}
```

Assembly

An assembly is **a collection of types and resources that are built to work together and form a logical unit of functionality**. Assemblies take the form of executable (.exe) or dynamic link library (. dll) files and they are a single unit of deployment of .NET applications.



Access Modifier

Access modifiers are keywords used to specify the declared accessibility of a field, method, constructor & class

Those access modifier allows you to grant and prevent access:

- public
- protected
- internal
- private
- protected-internal
- privare-protected

Access Modifier

Caller's location	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

https://code-maze.com/csharp-access-modifiers/

Recap

- Comment
- Variable and Data Type
  - int, long, float, double, enum, nullable, string
- Operator
- Flow Control
  - for, foreach, while
- Keywords
  - namespace, access modifier

Question?